Part FOUR
Streams and Collections

Chapter TWELVE
Streams

Exam Objectives

Describe Stream interface and Stream pipeline.
Use method references with Streams.

A simple example

Suppose you have a list of students and the requirements are to extract the students with a score of `90.0` or greater and sort them by score in ascending order.

One way to do it would be:

```
List<Student> studentsScore = new ArrayList<Student>();
for(Student s : students) {
    if(s.getScore() > 90.0) {
        studentsScore.add(s);
    }
}
Collections.sort(studentsScore, new Comparator<Student>() {
    public int compare(Student s1, Student s2) {
        return Double.compare(
           s1.getScore(), s2.getScore());
    }
});
```

Very verbose when we compare it with the Java 8 implementation using streams:

```
List<Student> studentsScore = students
        .stream()
        .filter(s -> s.getScore() >= 90.0)
        .sorted(Comparator.comparing(Student::getScore))
        .collect(Collectors.toList());
```

Don't worry if you don't fully understand the code; we'll see what it means later.

What are streams?

First, streams are **NOT** collections.

A simple definition is that streams are **WRAPPERS** for collections and arrays. They wrap an **EXISTING** collection (or another data source) to support operations expressed with **LAMBDAS**, so you specify what you want to do, not how to do it. You already saw it.

These are the characteristics of a stream:

Streams work perfectly with lambdas.
All streams operations take functional interfaces as arguments, so you can simplify the code with lambda expressions (and method references).

Streams don't store its elements.
The elements are stored in a collection or generated on the fly. They are only carried from the source through a pipeline of operations.

Streams are immutable.
Streams don't mutate their underlying source of elements. If, for example, an element is removed from the stream, a new stream with this element removed is created.

Streams are not reusable.
Streams can be traversed only once. After a terminal operation is executed (we'll see what this means in a moment), you have to create another stream from the source to further process it.

Streams don't support indexed access to their elements.
Again, streams are not collections or arrays. The most you can do is get their first element.

Streams are easily parallelizable.
With the call of a method (and following certain rules), you can make a stream execute its operations concurrently, without having to write any multithreading code.

Stream operations are lazy when possible.
Streams defer the execution of their operations either until the results are needed or until it's known how much data is needed.

One thing that allows this laziness is the way their operations are designed. Most of them return a new stream, allowing operations to be chained and form a pipeline that enables this kind of optimizations.

To set up this pipeline you:

1. Create the stream.
2. Apply zero or more intermediate operations to transform the initial stream into new streams.
3. Apply a terminal operation to generate a result or a "side-effect".

We're going to explain these steps and finally, we'll talk about more about laziness. In subsequent chapters, we'll go through all the operations supported by streams.

Creating streams

A stream is represented by the `java.util.stream.Stream<T>` interface. This works with objects only.

There are also specializations to work with primitive types, such as `IntStream`, `LongStream`, and `DoubleStream`.

There are many ways to create a stream. Let's start with the most popular three.

The first one is creating a stream from a `java.util.Collection` implementation using the `stream()` method:

```
List<String> words = Arrays.asList(new String[]{"hello", "hola", "hallo", "ciao"});
Stream<String> stream = words.stream();
```

The second one is creating a stream from individual values:

```
Stream<String> stream = Stream.of("hello","hola", "hallo", "ciao");
```

The third one is creating a stream from an array:

```
String[] words = {"hello", "hola", "hallo", "ciao"};
Stream<String> stream = Stream.of(words);
```

However, you have to be careful with this last method when working with primitives.

Here's why. Assume an `int` array:

```
int[] nums = {1, 2, 3, 4, 5};
```

When we create a stream from this array like this:

```
Stream.of(num)
```

We are not creating a stream of `Integer`s (`Stream<Integer>`), but a stream of `int` arrays (`Stream<int[]>`). This means that instead of having a stream with five elements we have a stream of one element:

```
System.out.println(Stream.of(nums).count()); // It prints 1!
```

The reason is the signatures of the of method:

```
// returns a stream of one element
static <T> Stream<T> of(T t)
// returns a stream whose elements are the specified values
static <T> Stream<T> of(T... values)
```

Since an int is not an object, but `int[]` is, the method chosen to create the stream is the first (`Stream.of(T t)`), not the one with the vargs, so a stream of `int[]` is created, but since only one array is passed, the result is a stream of one element.

To solve this, we can force Java to choose the vargs version by creating an array of objects (with `Integer`):

```
Integer[] nums = {1, 2, 3, 4, 5};
// It prints 5!
System.out.println(Stream.of(nums).count());
```

Or use a fourth way to create a stream (that it's in fact used inside `Stream.of(T... values)`):

```
int[] nums = {1, 2, 3, 4, 5};
// It also prints 5!
System.out.println(Arrays.stream(nums).count());
```

Or use the primitive version `IntStream`:

```
int[] nums = {1, 2, 3, 4, 5};
// It also prints 5!
System.out.println(IntStream.of(nums).count());
```

Lesson learned: Don't use `Stream<T>.of()` when working with primitives.

Here are other ways to create streams.

```
static <T> Stream<T> generate(Supplier<T> s)
```

This method returns an "infinite" stream where each element is generated by the provided `Supplier`, and it's generally used with the method:

```
Stream<T> limit(long maxSize)
```

That truncates the stream to be no longer than `maxSize` in length.

For example:

```
Stream<Double> s = Stream.generate(new Supplier<Double>() {
    public Double get() {
        return Math.random();
    }
}).limit(5);
```

Or:

```
Stream<Double> s = Stream.generate(() -> Math.random()).limit(5);
```

Or just:

```
Stream<Double> s = Stream.generate(Math::random).limit(5);
```

That generates a stream of five random `double`s.

```
static <T> Stream<T> iterate(T seed, UnaryOperator<T> f)
```

This method returns an "infinite" stream produced by the iterative application of a function `f` to an initial element seed. The first element (`n = 0`) in the stream will be the provided seed. For `n > 0`, the element at position `n` will be the result of applying the function `f` to the element at position `n - 1`. For example:

```
Stream<Integer> s = Stream.iterate(1, new UnaryOperator<Integer>() {
    @Override
    public Integer apply(Integer t) {
        return t * 2; }
}).limit(5);
```

Or just:

```
Stream<Integer> s = Stream.iterate(1, t -> t * 2).limit(5);
```

That generates the elements `1`, `2`, `4`, `8`, `16`.

There's a `Stream.Builder<T>` class (that follows the builder design pattern) with methods that add an element to the stream being built:

```
void accept(T t) Stream.Builder<T> add(T t)
```

For example:

```
Stream.Builder<String> builder =
    Stream.<String>builder()
       .add("h").add("e").add("l").add("l");
builder.accept("o");
Stream<String> s = builder.build();
```

`IntStream` and `LongStream` define the methods:

```
static IntStream range(long startInclusive, long endExclusive)
static IntStream rangeClosed(long startInclusive, long endInclusive)
static LongStream range(long startInclusive, long endExclusive)
static LongStream rangeClosed(long startInclusive, long endInclusive)
```

That returns a sequential stream for the range of `int` or `long` elements. For example:

```
// stream of 1, 2, 3
IntStream s = IntStream.range(1, 4);
// stream of 1, 2, 3, 4
IntStream s = IntStream.rangeClosed(1, 4);
```

Also, there are new methods in the Java API that generate streams. For example:

```
IntStream s1 = new Random().ints(5, 1, 10);
```

That returns an `IntStream` of five random `int` s from one (inclusive) to ten (exclusive).

Intermediate operations

You can easily identify intermediate operations; they always return a new stream. This allows the operations to be connected.

For example:

```
Stream<String> s = Stream.of("m", "k", "c", "t")
    .sorted()
    .limit(3)
```

An important feature of intermediate operations is that they don't process the elements until a terminal operation is invoked, in other words, they're lazy.

Intermediate operations are further divided into *stateless* and *stateful* operations.

Stateless operations retain no state from previously elements when processing a new element so each can be processed independently of operations on other elements.

Stateful operations, such as distinct and sorted, may incorporate state from previously seen elements when processing new elements.

The following table summarizes the methods of the `Stream` interface that represent intermediate operations.

Method	Type	Description
`Stream<T> distinct()`	Stateful	Returns a stream consisting of the distinct elements.
`Stream<T> filter(Predicate<? super T> predicate)`	Stateless	Returns a stream of elements that match the given predicate.
`<R> Stream<R> flatMap(Function<? super T,? extends Stream<? extends R>> mapper)`	Stateless	Returns a stream with the content produced by applying the provided mapping function to each element. There are versions for `int`, `long` and `double` also.
`Stream<T> limit(long maxSize)`	Stateful	Returns a stream truncated to be no longer than `maxSize` in length.
`<R> Stream<R> map(Function<? super T,? extends R> mapper)`	Stateless	Returns a stream consisting of the results of applying the given function to the elements of this stream. There are versions for `int`, `long` and `double` also.
`Stream<T> peek(Consumer<? super T> action)`	Stateless	Returns a stream with the elements of this stream, performing the provided action on each element.
`Stream<T> skip(long n)`	Stateful	Returns a stream with the remaining elements of this stream after discarding the first n elements.
`Stream<T> sorted()`	Stateful	Returns a stream sorted according to the natural order of its elements.
`Stream<T> sorted(Comparator<? super T> comparator)`	Stateful	Returns a stream with the sorted according to the provided `Comparator`.
`Stream<T> parallel()`	N/A	Returns an equivalent stream that is parallel.
`Stream<T> sequential()`	N/A	Returns an equivalent stream that is sequential.
`Stream<T> unordered()`	N/A	Returns an equivalent stream that is unordered.

Terminal operations

You can also easily identify terminal operations, they always return something other than a stream.

After the terminal operation is performed, the stream pipeline is *consumed*, and can't be used anymore. For example:

```
int[] digits = {0, 1, 2, 3, 4 , 5, 6, 7, 8, 9};
IntStream s = IntStream.of(digits);
long n = s.count();
System.out.println(s.findFirst()); // An exception is thrown
```

If you need to traverse the same stream again, you must return to the data source to get a new one. For example:

```
int[] digits = {0, 1, 2, 3, 4 , 5, 6, 7, 8, 9};
long n = IntStream.of(digits).count();
System.out.println(IntStream.of(digits).findFirst()); // OK
```

The following table summarizes the methods of the `Stream` interface that represent terminal operations.

Method	Description
`boolean allMatch(Predicate<? super T> predicate)`	Returns whether all elements of this stream match the provided predicate.
`boolean anyMatch(Predicate<? super T> predicate)`	Returns whether any elements of this stream match the provided predicate.
`boolean noneMatch(Predicate<? super T> predicate)`	Returns whether no elements of this stream match the provided predicate.
`Optional<T> findAny()`	Returns an `Optional` describing some element of the stream.
`Optional<T> findFirst()`	Returns an `Optional` describing the first element of this stream.
`<R,A> R collect(Collector<? super T,A,R> collector)`	Performs a mutable reduction operation on the elements of this stream using a `Collector` .
`long count()`	Returns the count of elements in this stream.
`void forEach(Consumer<? super T> action)`	Performs an action for each element of this stream.
`void forEachOrdered(Consumer<? super T> action)`	Performs an action for each element of this stream, in the encounter order of the stream if the stream has a defined encounter order.
`Optional<T> max(Comparator<? super T> comparator)`	Returns the maximum element of this stream according to the provided `Comparator` .
`Optional<T> min(Comparator<? super T> comparator)`	Returns the maximum element of this stream according to the provided `Comparator` .
`T reduce(T identity, BinaryOperator<T> accumulator)`	Performs a reduction on the elements of this stream, using the provided identity value and an associative accumulation function, and returns the reduced value.
`Object[] toArray()`	Returns an array containing the elements of this stream.
`<A> A[] toArray(IntFunction<A[]> generator)`	Returns an array containing the elements of this stream, using the provided generator function to allocate the returned array.
`Iterator<T> iterator()`	Returns an iterator for the elements of the stream.
`Spliterator<T> spliterator()`	Returns a spliterator for the elements of the stream.

Lazy operations

Intermediate operations are deferred until a terminal operation is invoked. The reason is that intermediate operations can usually be merged or optimized by a terminal operation.

Let's take for example this stream pipeline:

```
Stream.of("sun", "pool", "beach", "kid", "island", "sea", "sand")
    .map(str -> str.length())
    .filter(i -> i > 3)
    .limit(2)
    .forEach(System.out::println);
```

Here's what it does:

- It generates a stream of strings,
- Then convert the stream to a stream of `int` s (representing the length of each string)
- Then it filters the lengths greater than three,
- Then it grabs the first two elements of the string and
- Finally, prints those two elements.

And you may think the map operation is applied to all seven elements, then the filter operation again to all seven, then it picks the first two, and finally it prints the values.

But this is not how it works. If we modify the lambda expressions of map and filter to print a message:

```
Stream.of("sun", "pool", "beach", "kid", "island", "sea", "sand")
    .map(str -> {
        System.out.println("Mapping: " + str);
        return str.length();
    })
    .filter(i -> {
        System.out.println("Filtering: " + i);
        return i > 3;
    })
    .limit(2)
    .forEach(System.out::println);
```

The order of evaluation will be revealed:

```
Mapping: sun
Filtering: 3
Mapping: pool
Filtering: 4
4
Mapping: beach
Filtering: 5
5
```

From this example, we can see the stream didn't apply all the operations on the pipeline to all elements, only until if it finds the elements needed to return a result (due to the `limit(2)` operation). This is called *short-circuiting*.

Short-circuit operations cause intermediate operations to be processed until a result can be produced.

In such a way, because of lazy and short-circuit operations, streams don't execute all operations on all their elements. Instead, the elements of the stream go through a pipeline of operations until the point a result can be deduced or generated.

You can see short-circuiting as a subclassification. There's only one short-circuit intermediate operation, while the rest are terminal:

INTERMEDIATE

```
Stream<T> limit(long maxSize)
```

(Because it doesn't need to process all the elements of the stream to create a stream of a given size)

TERMINAL

```
boolean anyMatch(Predicate<? super T> predicate)
boolean allMatch(Predicate<? super T> predicate)
boolean noneMatch(Predicate<? super T> predicate)
Optional<T> findFirst()
Optional<T> findAny()
```

(Because as soon as you find a matching element, there's no need to continuing processing the stream)

In the next chapters, we'll review the rest of the operations of the `Stream` interface.

Key Points

- Streams can be defined as wrappers for collections and arrays. They wrap an existing collection (or another data source) to support operations expressed with lambdas, so you specify what you want to do, not how to do it.
- These are the characteristics of a stream:
  - Streams work perfectly with lambdas.
  - Streams don't store its elements.
  - Streams are immutable.
  - Streams are not reusable.
  - Streams don't support indexed access to their elements.
  - Streams are easily parallelizable.
  - Stream operations are lazy when possible.
- Stream operations can be chained to form a pipeline. To set up this pipeline you:
  1. Create the stream.
  2. Apply zero or more intermediate operations to transform the initial stream into new streams.
  3. Apply a terminal operation to generate a result or a "side-effect".
- There are many ways to create a stream. The most popular are:

```
// From an existing collection
List<String> words = Arrays.asList(new String[]{"hello", "hola", "hallo", "ciao"}
Stream<String> s1 = words.stream();

// From individual elements
Stream<String> s2 = Stream.of("hello", "hola", "hallo", "ciao");

// From an array
String[] words = {"hello", "hola", "hallo", "ciao"};
Stream<String> s3 = Stream.of(words);
```

- Don't use `Stream<T>.of()` when working with primitives. Instead use `Arrays.stream` or the primitive versions of `Stream`:

```
int[] nums = {1, 2, 3, 4, 5};
IntStream s1 = Arrays.stream(nums);
IntStream s2 = IntStream.of(nums);
```

- Intermediate operations such as `map` or `filter` always return a new stream and are divided into *stateless* and *stateful* operations, and they are lazy, meaning that they are deferred until a terminal operation is invoked.

- Terminal operations such as `count` or `forEach` always return something other than a stream.
- Short-circuit operations cause intermediate operations to be processed until a result can be produced.
- In such a way, because of lazy and short-circuit operations, streams don't execute all operations on all their elements, but until the point a result can be deduced or generated.

Self Test

1. Given:

```
public class Question_12_1 {
    public static void main(String[] args) {
        IntStream.range(1, 10)
            .filter(i -> {
                System.out.print("1");
                return i % 2 == 0;
            })
            .filter(i -> {
                System.out.print("0");
                return i > 3;
            })
            .limit(1)
            .forEach(i -> {
                System.out.print(i);
            });
    }
}
```

What is the result?
A. `101010104`
B. `1111111110000000004`
C. `11041106`
D. `1101104`
E. An exception is thrown

2. Which of the following are intermediate operations?
A. `limit`
B. `peek`
C. `anyMatch`
D. `skip`

3. Which of the following are terminal operations?
A. `sorted`
B. `flatMap`
C. `max`
D. `distinct`

4. Which of the following are short-circuit operations?
A. `reduce`
B. `parallel`
C. `findNone`
D. `findFirst`

5. Given:

```
public class Question_12_2 {
    public static void main(String[] args) {
        IntStream.range(1, 5).count().limit(4);
```

```
    }
}
```

What is the result?
A. 5
B. 4
C. 1
D. Compilation error
E. An exception is thrown

Open answers page

Do you like what you read? Would you consider?

Buying the print/kindle version from Amazon

Buying the PDF/EPUB/MOBI versions from Leanpub

Buying the e-book version from iTunes

Buying the e-book version from Kobo

Buying the e-book version from Scribd

Do you have a problem or something to say?

Report an issue with the book

Contact me

---

11. Method References 13. Iterating and Filtering Collections